Java Concurrency

Harshit Bangar

Credit - Douglas Hawkins

Quiz - Identify thread safe APIs

- long height = 64L
- x += 1
- Point point = new Point();

Quiz - Identify thread safe APIs

- long height = 64L
- x += 1
- Point point = new Point();

Quiz - Identify thread safe APIs

- long height = 64L
- x += 1
- Point point = new Point();

Quiz - Identify thread safe APIs

- long height = 64L
- x += 1
- Point point = new Point();

Content

Content

- Unit testing concurrent code.

Content

- Unit testing concurrent code.
- Atomicity

Content

- Unit testing concurrent code.
- Atomicity
- Visibility

Content

- Unit testing concurrent code.
- Atomicity
- Visibility
- Thread Safety

Content

- Unit testing concurrent code.
- Atomicity
- Visibility
- Thread Safety
- Designing a thread safe class

Content

- Unit testing concurrent code.
- Atomicity
- Visibility
- Thread Safety
- Designing a thread safe class
- Java Memory Model

Unit testing

Unit testing

1. The biggest challenge with testing concurrent code is unpredictability. Unlike code which is sequential and deterministic, concurrency can yield completely random results

Unit testing

1. The biggest challenge with testing concurrent code is unpredictability. Unlike code which is sequential and deterministic, concurrency can yield completely random results
2. Unit tests sometimes introduces sequentiality by accident

Unit testing

Unit testing

- Junit

Unit testing

- Junit
- TestNg

Unit testing

- Junit
- TestNg
- ThreadWeaver

Unit testing

- Junit
- TestNg
- ThreadWeaver
- JCStress

Unit testing - Junit & TestNg

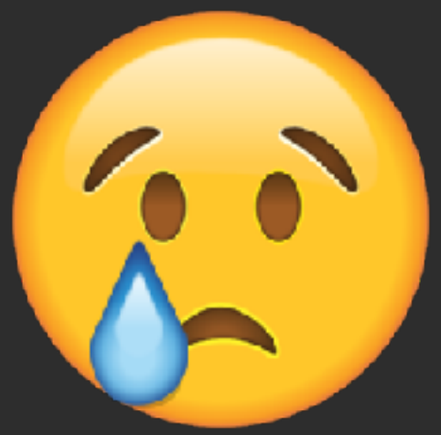
Unit testing - Junit & TestNg

- Junit doesn’t support concurrent testing.

Unit testing - Junit & TestNg

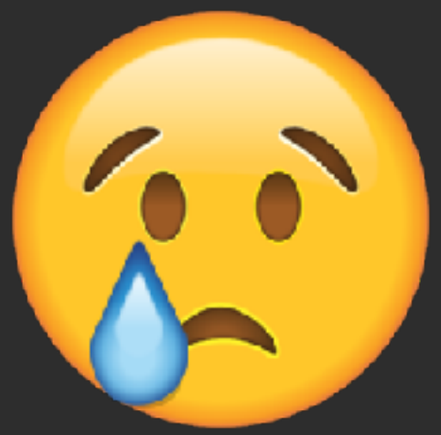
- Junit doesn’t support concurrent testing.
- TestNg supports testing but it is broken since it is unable to reproduce the consistent behaviour.

Unit testing - Thread Weaver



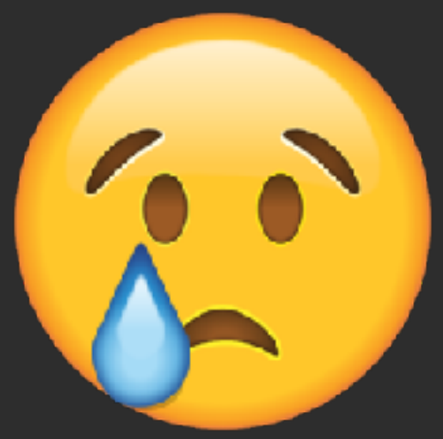
Unit testing - Thread Weaver

- Works by adding conditional breakpoints in the code (by inserting locks via byte code manipulation)



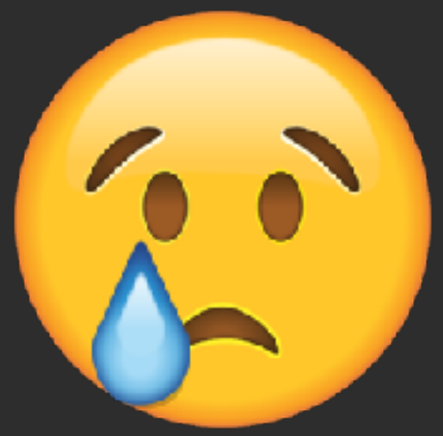
Unit testing - Thread Weaver

- Works by adding conditional breakpoints in the code (by inserting locks via byte code manipulation)
- Theoretically should catch every race condition.



Unit testing - Thread Weaver

- Works by adding conditional breakpoints in the code (by inserting locks via byte code manipulation)
- Theoretically should catch every race condition.
- It prevent JMM re-ordering (accidentally) due to locks - details.



Unit testing - JCStress

Unit testing - JCStress

- Stress testing by simulating concurrency via multiple actors

Unit testing - JCStress

- Stress testing by simulating concurrency via multiple actors
- Not a unit test, result might vary from run to run.

Unit testing - JCStress

- Stress testing by simulating concurrency via multiple actors
- Not a unit test, result might vary from run to run.
- Result might depend on the processor architecture.

Unit testing - JCStress

- Stress testing by simulating concurrency via multiple actors
- Not a unit test, result might vary from run to run.
- Result might depend on the processor architecture.
- Examples

ATOMICITY

WHAT OPERATIONS ARE INDIVISIBLE?

SUCCINCT != ATOMIC

```
sharedX = 2L;
```

thread 1	thread 2
sharedX = 2L	sharedX = -1L
set_hi sharedX, 0000 0000	set_hi sharedX, ffff ffff
set_lo sharedX, 0000 0002	set_lo sharedX, ffff ffff

JLS8 § 17.7 pp 6

SUCCINCT != ATOMIC

```
sharedX += 1;
```

```
localX = sharedX;
localX = localX + 1;
sharedX = localX;
```

SUCCINCT != ATOMIC

```
Point sharedPoint = new Point(x, y);
```

```
local1 = calloc(sizeof(Point));
local1.<init>(x, y);
  Object.<init>();
  this.x = x;
  this.y = y;
sharedPoint = local1;
```

VISIBILITY

WHAT CAN OTHER THREADS SEE?

NO GARBAGE VALUES

NO OUT OF "THIN AIR" VALUES

WILL SEE A ZERO-ISH VALUE

OR

AN ASSIGNED VALUE

NO *COMPLETELY* GARBAGE VALUES

TWO SEPARATE HALVES OF LONG OR DOUBLE

BUT

BOTH HALVES ARE ZERO OR ASSIGNED

ONLY ASSIGNED VALUES FOR FINALS*

```
sharedPoint = new Point(x, y);
```

```
class Point {
  final int x, y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

JLS8 § 17.5 pp 652

*After constructor return

Clip slide

NOT *ALL* VALUES ARE STORED

```
sharedX = 20;
```

← Dead Store

```
sharedX = 40;

print(sharedX);
```

DATA DEPENDENCE

```
sharedX = 20;
```
← Dead Store

Data Dependence

```
sharedX = 40;

print(sharedX);
```

Clip slide

DATA DEPENDENCE

```
sharedX = 20;
sharedX = 40;
print(40);
```

Dead Stores

Data Dependence

Propagate Data

NOT *ALL* VALUES ARE READ

```
xSquared = sharedX * sharedX
```

Can xSquared be 30?

```
local1 = sharedX
local2 = sharedX
xSquared = local1 * local2
```

NOT *ALL* VALUES ARE READ

Can xSquared be 30?

```
local1 = sharedX
local2 = sharedX
xSquared = local1 * local2
```

thread 1	thread 2
	sharedX = 5;
local1 = sharedX; (5)	
	sharedX = 6;
local2 = sharedX; (6)	
xSquared = local1 * local2; (30)	

Clip slide

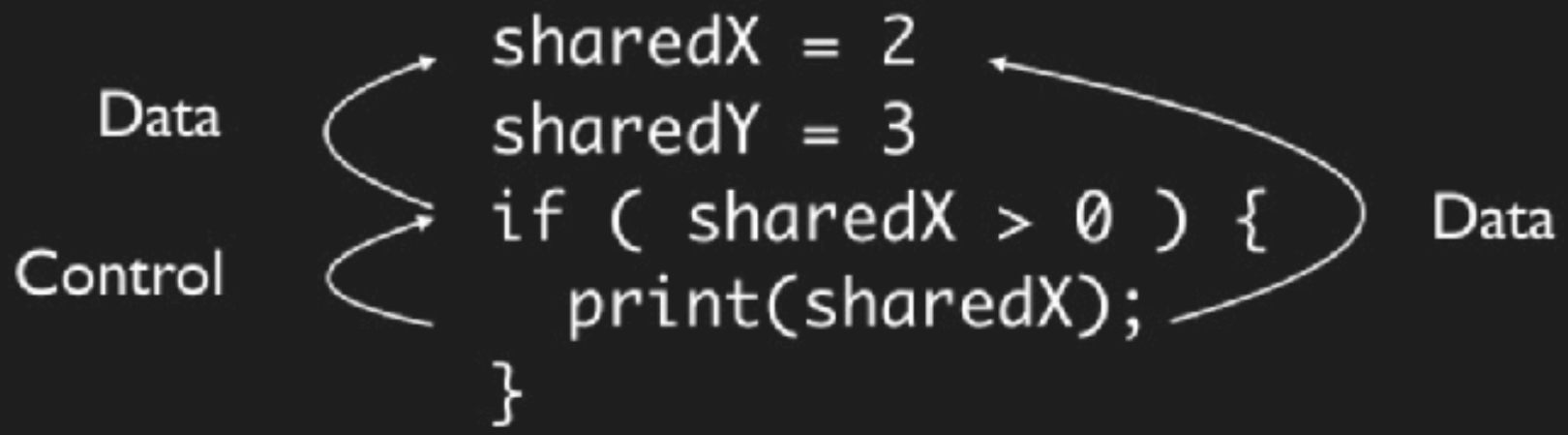
ORDERING

WHAT ORDER DOES IT RUN IN?

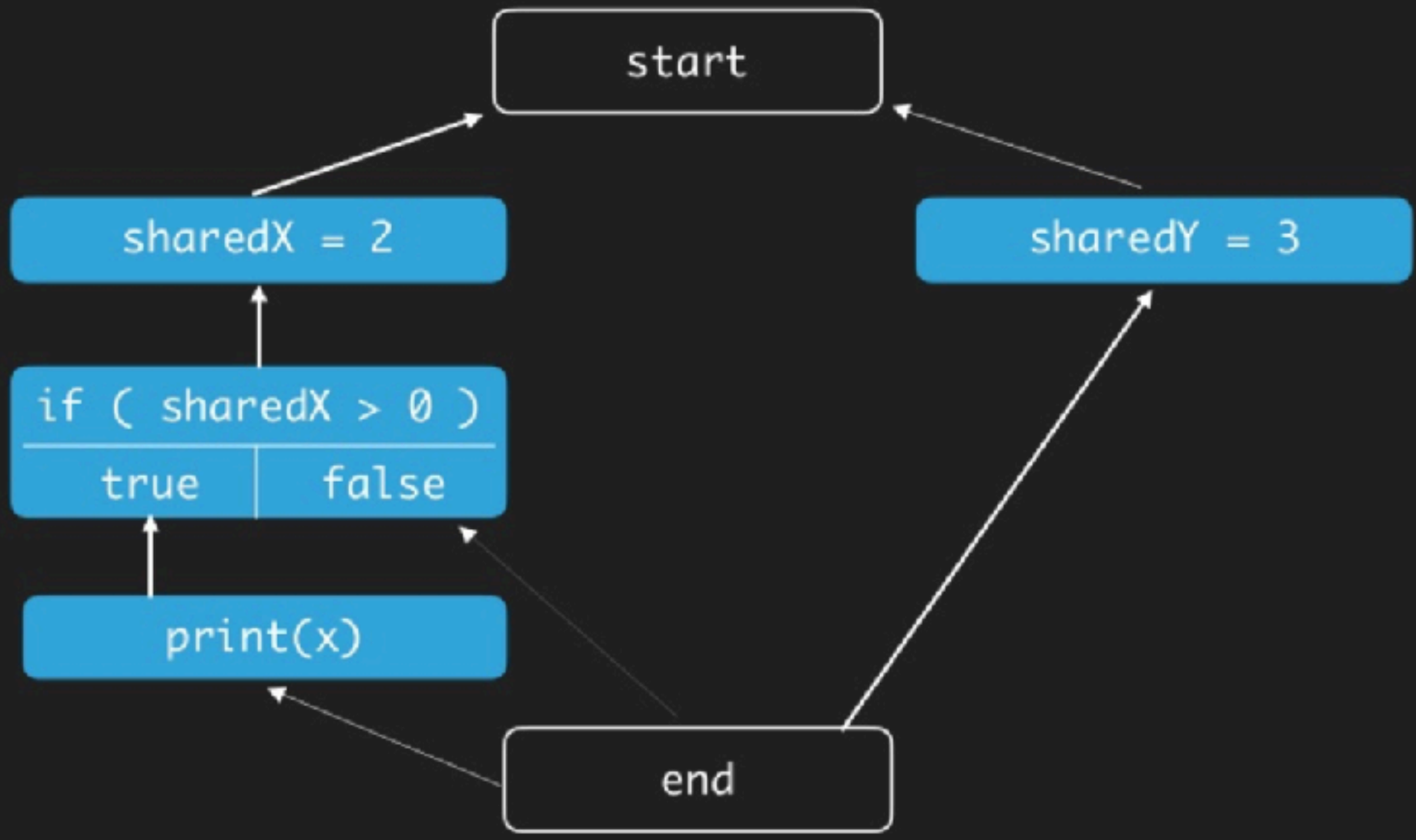
Clip slide

"PROGRAM ORDER"

DATA + CONTROL DEPENDENCE

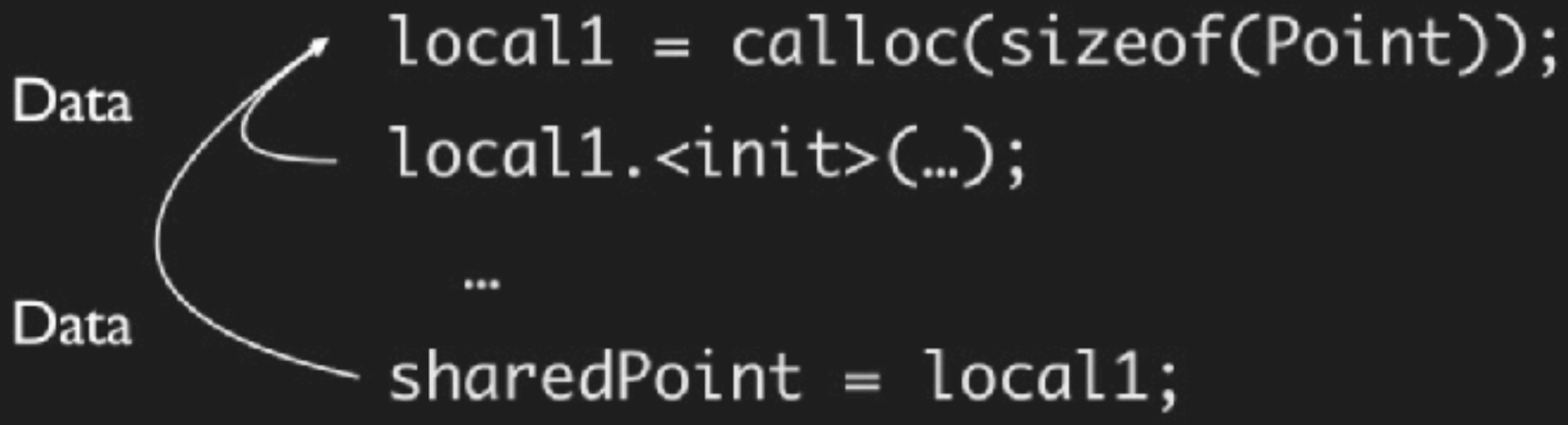


DATA + CONTROL DEPENDENCE



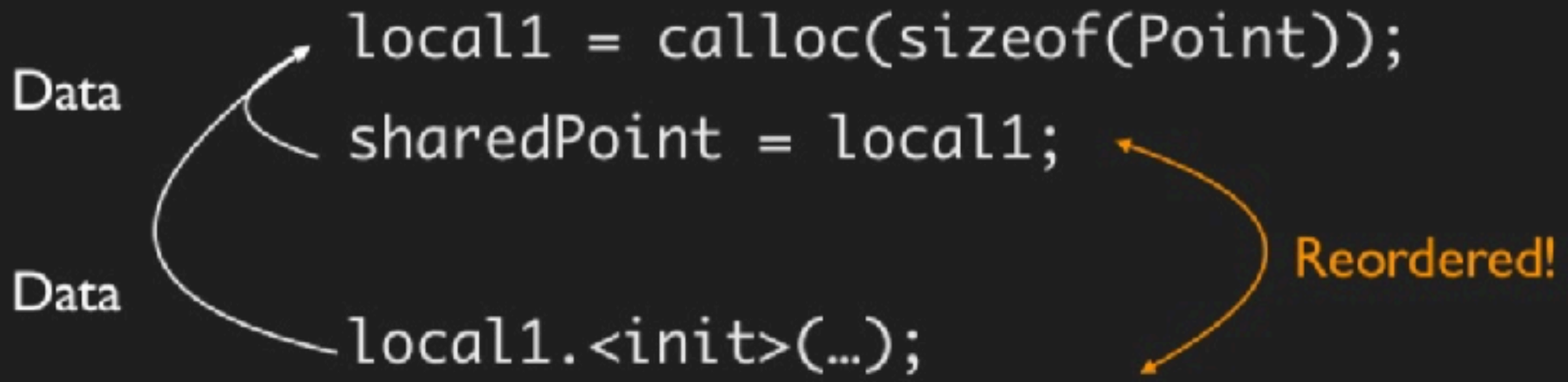
STORE SOONER

```
sharedFoo = new Foo(…);
```



FREE TO REORDER

```
sharedFoo = new Foo(…);
```



BROKEN DOUBLE CHECKED LOCKING

```
static Singleton instance() {
 if ( sharedInstance == null ) {
  synchronized ( Singleton.class ) {
   if ( sharedInstance == null ) {
    sharedInstance = new Singleton();
   }
  }
 }
 return sharedInstance;
}
```

```
static Singleton instance() {
 if ( sharedInstance == null ) {
  synchronized ( Singleton.class ) {
   if ( sharedInstance == null ) {
    local = calloc(sizeof(Singleton));
    sharedInstance = local;
    local.<init>();
   }
  }
 }
 return sharedInstance;
}
```

sharedInstance is non-null,
but constructor hasn't run.

PRODUCER

```
...
sharedData = ...;
sharedDone = true;
```

Reorder!

```
...
sharedDone = true;
sharedData = ...;
```

CONSUMER

```
while ( !sharedDone );
print(sharedData);
```

Clip slide

AGAIN BLAME HARDWARE.

Clip slide

	RISC / ARM	CISC / x86
Loads After Loads	YES	YES
Loads After Stores	YES	YES
Stores After Stores	YES	NO
Stores After Loads	YES	YES

TO STOP REORDERING, USE A *FENCE*

PRODUCER

```
...
sharedData = …;
--- store_store_fence(); ---
sharedDone = true;
```

CONSUMER

```
while ( !sharedDone ) {
--- load_load_fence(); ---
}
print(sharedData);
```

Clip slide

BLAME JAVA,

TOO.

	RISC / ARM	CISC / x86	Java
Loads After Loads	YES	YES	YES
Loads After Stores	YES	YES	YES
Stores After Stores	YES	NO	YES
Stores After Loads	YES	YES	YES

HOW DO YOU TELL JAVA TO NOT REORDER?

SYNCHRONIZATION ACTIONS

VOLATILE

SYNCHRONIZED

FINAL / FREEZE

~~UNSAFE~~

ATOMICS

VAR HANDLES

FINAL & FREEZE

```
Point sharedPoint = new Point(x, y);

        local1 = calloc(sizeof(Point));
local1.<init>(x, y);
  Object.<init>();
  this.x = x;
  this.y = y;
  freeze();
sharedPoint = local1;
```

FINAL & FREEZE

```
Point sharedPoint = new Point(x, y);

        local1 = calloc(sizeof(Point));
      local1.<init>(x, y);
        Object.<init>();
        this.x = x;
        this.y = y;
        OtherThread.see(this);
      ---------- freeze(); ----------
      sharedPoint = local1;
```

Free to Reorder!

Final & Freeze

```
class FinalWrapperFactory {
  private FinalWrapper w;

  public Singleton get() {
    if (w == null) { // check 1
      synchronized(this) {
        if (w == null) { // check2
          w = new FinalWrapper(new Singleton());
        }
      }
      return w.instance;
    }

  private static class FinalWrapper {
    public final Singleton instance;
    public FinalWrapper(Singleton instance) {
      this.instance = instance;
    }
  }
}
```

Final & Freeze

```
class FinalWrapperFactory {
  private FinalWrapper w;

  public Singleton get() {
    if (w == null) { // check 1
      synchronized(this) {
        if (w == null) { // check2
          w = new FinalWrapper(new Singleton());
        }
      }
    return w.instance;
  }

  private static class FinalWrapper {
    public final Singleton instance;
    public FinalWrapper(Singleton instance) {
      this.instance = instance;
    }
  }
}
```

Final & Freeze

```
class FinalWrapperFactory {
  private FinalWrapper w;

  public Singleton get() {
    if (w == null) { // check 1
      synchronized(this) {
        if (w == null) { // check2
          w = new FinalWrapper(new Singleton());
        }
      }
    }
    return w.instance;
  }

  private static class FinalWrapper {
    public final Singleton instance;
    public FinalWrapper(Singleton instance) {
      this.instance = instance;
    }
  }
}
```

Final & Freeze

```
class FinalWrapperFactory {
  private FinalWrapper w;

  public Singleton get() {
    if (w == null) { // check 1
      synchronized(this) {
        if (w == null) { // check2
          w = new FinalWrapper(new Singleton());
        }
      }
      return w.instance;
    }

  private static class FinalWrapper {
    public final Singleton instance;
    public FinalWrapper(Singleton instance) {
      this.instance = instance;
    }
  }
}
```

VOLATILE

PRODUCER

```
...
volatileData = …;
volatileDone = true;
```

CONSUMER

```
while ( !volatileDone ) {
}
print(volatileData);
```

ALSO FIXES DOUBLE CHECKED LOCKING

```
static Singleton instance() {
 if ( sharedInstance == null ) {
  synchronized ( Singleton.class ) {
   if ( sharedInstance == null ) {
    local = calloc(sizeof(Singleton));
    local.<init>();
    sharedInstance = local;
   }
  }
 }
 return sharedInstance;
}
```

RIGHT WAY TO DO LAZY SINGLETON IN JAVA

CLASS INITIALIZATION IS ALREADY LAZY.

```
static Singleton instance() {
 return Holder.INSTANCE;
}

static class Holder {
  static final class Singleton INSTANCE = new Singleton();
}
```

Mutable state

There are three ways to fix it:

Mutable state

There are three ways to fix it:

- Use synchronisation whenever accessing the state variable.

Mutable state

There are three ways to fix it:

- Use synchronisation whenever accessing the state variable.
- Don't share the state variable across threads;

Mutable state

There are three ways to fix it:

- Use synchronisation whenever accessing the state variable.
- Don't share the state variable across threads;
- Make the state variable immutable; or

Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Factorizer

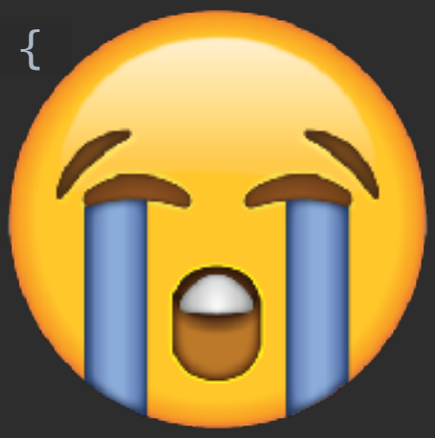
```
public class Factorizer extends GenericServlet implements Servlet {

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {

  private long count = 0;

  public long getCount() {
    return count;
  }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    ++count;
    encodeIntoResponse(resp, factors);
  }
}
```

Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {

  private long count = 0;

  public long getCount() {
    return count;
  }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    ++count;
    encodeIntoResponse(resp, factors);
  }
}
```



Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {

  private final AtomicLong count = new AtomicLong(0);

  public long getCount() {
    return count;
  }

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    count.incrementAndGet();
    encodeIntoResponse(resp, factors);
  }
}
```

COMPARE & SWAP

```
AtomicInteger atomic = new AtomicInteger(0);

atomic.getAndIncrement();
```

```
boolean applied = false;
do {
    int value = sharedVar.get();
    applied = sharedVar.compareAndSet(
      value, value + 1);
} while ( ! applied );
```

Concurrent Stack

```
public class ConcurrentStack <E> {
  AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = top.get();
      newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead));
  }

  public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
  }
}
```

Concurrent Stack

```
public class ConcurrentStack <E> {
  AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = top.get();
      newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead));
  }

  public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
  }
}
```

Concurrent Stack

```
public class ConcurrentStack <E> {
  AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

  public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
      oldHead = top.get();
      newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead));
  }

  public E pop() {
    Node<E> oldHead;
    Node<E> newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
  }
}
```

Factorizer

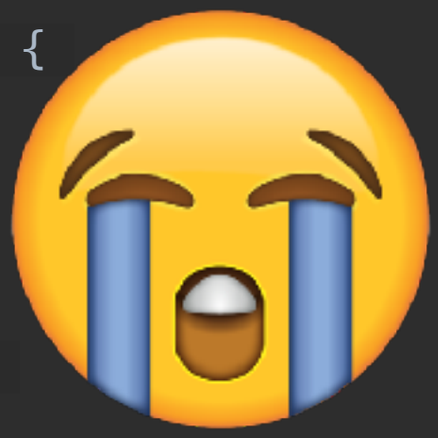
```
public class Factorizer extends GenericServlet implements Servlet {

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    encodeIntoResponse(resp, factors);
  }
}
```

Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
      = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
      = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {
  private final AtomicReference<BigInteger> lastNumber
      = new AtomicReference<BigInteger>();
  private final AtomicReference<BigInteger[]> lastFactors
      = new AtomicReference<BigInteger[]>();

  public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    if (i.equals(lastNumber.get()))
      encodeIntoResponse(resp, lastFactors.get());
    else {
      BigInteger[] factors = factor(i);
      lastNumber.set(i);
      lastFactors.set(factors);
      encodeIntoResponse(resp, factors);
    }
  }
}
```

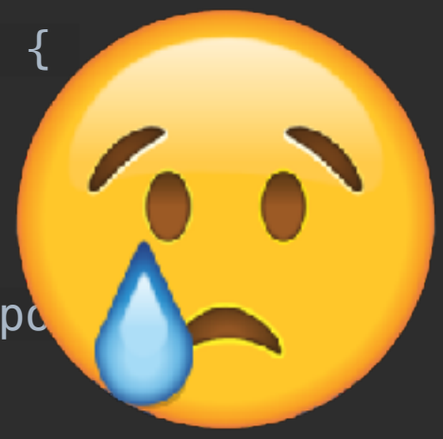


“To preserve state consistency, update related state variables in a **single atomic operation**.”

–Commandment #1

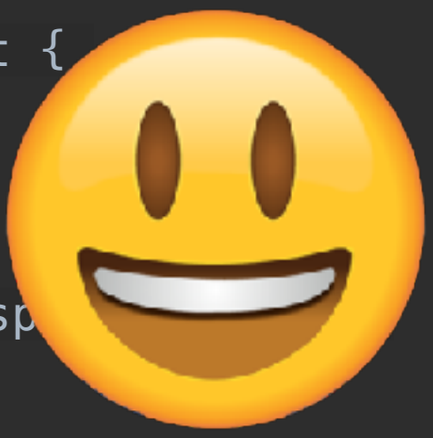
Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber;
  @GuardedBy("this") private BigInteger[] lastFactors
      = new BigInteger[]{};

  public synchronized void service(ServletRequest req, ServletRespo
    BigInteger i = extractFromRequest(req);
    if (i == lastNumber)
      encodeIntoResponse(resp, lastFactors);
    else {
      BigInteger[] factors = factor(i);
      lastNumber = i;
      lastFactors = factors;
      encodeIntoResponse(resp, factors);
    }
  }
}
```



Factorizer

```
public class Factorizer extends GenericServlet implements Servlet {
  @GuardedBy("this") private BigInteger lastNumber;
  @GuardedBy("this") private BigInteger[] lastFactors
      = new BigInteger[]{};

  public synchronized void service(ServletRequest req, ServletResp
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = null;
    synchronized (this) {
      if (i.equals(lastNumber)) {
        factors = lastFactors.clone();
      }
    }
    if (factors == null) {
      factors = factor(i);
      synchronized (this) {
        lastNumber = i;
        lastFactors = factors.clone();
      }
    }
    encodeIntoResponse(resp, factors);
  }
}
```



Synchronized

```
public class SynchronizedDemo {

  public synchronized void update() { }
}
```

Synchronized

```
public class SynchronizedDemo {

  public synchronized void update() {
    synchronized (this) {

    }
  }
}
```

Synchronized

```
public class SynchronizedDemo {
  private Object lock = new Object();
  public synchronized void update() {
    synchronized (lock) {

    }
  }
}
```

Synchronized

```
public class SynchronizedDemo {
  @GuardedBy("lock") Integer counter1;
  private Object lock = new Object();
  public void update() {
    synchronized (lock) {
      if (counter1 > Integer.MAX_VALUE) {
        return;
      }
      counter1 = counter1 + 1;
    }
  }
}
```

Synchronized

```
public class SynchronizedDemo {
  @GuardedBy("lock") Integer counter1;
  @GuardedBy("lock2") Integer counter2;
  private Object lock = new Object();
  private Object lock2 = new Object();
  public void update() {
  }
  public void update2() {
    synchronized (lock2) {
      if (counter2 > Integer.MAX_VALUE) {
        return;
      }
      counter2 = counter2 + 1;
    }
  }
}
```

Synchronized

```
public class SynchronizedDemo extends BaseUpdate {

  public synchronized void update() {
    super.update();
  }
}
```

```
class BaseUpdate {
  public synchronized void update() { }
}
```

Locking

Locking

- Prefer explicit locking. Exception - superclass uses synchronised keyword, private lock will be unusable for subclasses

Locking

- Prefer explicit locking. Exception - superclass uses synchronised keyword, private lock will be unusable for subclasses
- Locks are reentrant.

Locking

- Prefer explicit locking. Exception - superclass uses synchronised keyword, private lock will be unusable for subclasses
- Locks are reentrant.
- Add annotation @GuardedBy(“CorrectLock”) for documentation.

Locking

- Prefer explicit locking. Exception - superclass uses synchronised keyword, private lock will be unusable for subclasses
- Locks are reentrant.
- Add annotation @GuardedBy(“CorrectLock”) for documentation.
- Perform all operations on a single variable using the same lock.

Locking

- Prefer explicit locking. Exception - superclass uses synchronised keyword, private lock will be unusable for subclasses
- Locks are reentrant.
- Add annotation @GuardedBy(“CorrectLock”) for documentation.
- Perform all operations on a single variable using the same lock.
- Perform all operations on related variables using the same lock. (i.e. Unrelated variables can have different locks)

LOCK COARSENING

```
synchronized ( buffer ) {
  buffer.add(x);
}
foo = bar;
synchronized ( buffer ) {
  buffer.add(y);
}
```

```
synchronized ( buffer ) {
  buffer.add(x);
  foo = bar;
  buffer.add(y);
}
```

https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Thread Confinement

```
BehaviorSubject<String> timerSubject =
BehaviorSubject.create();

Observable
    .timer(4, TimeUnit.SECONDS)
    .subscribe(aLong -> {
      timerSubject.onNext(String.valueOf(aLong));
    });
```

Thread Confinement

```
BehaviorSubject<String> timerSubject =
BehaviorSubject.create();

Observable
    .timer(4, TimeUnit.SECONDS, Schedulers.computation())
    .subscribe(aLong -> {
      timerSubject.onNext(String.valueOf(aLong));
    });
```

```
* <p>All methods except the onSubscribe, onNext, onError and
onComplete are thread-safe.
* Use {@link #toSerialized()} to make these methods thread-
safe as well.
```

Thread Confinement

```
Subject<String> timerSubject =
BehaviorSubject.create().serialized();

Observable
    .timer(4, TimeUnit.SECONDS, Schedulers.computation())
    .subscribe(aLong -> {
      timerSubject.onNext(String.valueOf(aLong));
    });

* <p>All methods except the onSubscribe, onNext, onError and
onComplete are thread-safe.
* Use {@link #toSerialized()} to make these methods thread-
safe as well.
```

Thread Confinement

```
Scheduler scheduler = Schedulers.single();
BehaviorSubject<String> timerSubject
= BehaviorSubject.create();

Observable
    .timer(4, TimeUnit.SECONDS, Schedulers.computation())
    .observeOn(scheduler)
    .subscribe(aLong -> {
      timerSubject.onNext(String.valueOf(aLong));
    });

* <p>All methods except the onSubscribe, onNext, onError and
onComplete are thread-safe.
* Use {@link #toSerialized()} to make these methods thread-
safe as well.
```

RX-Java Thread Safety

RX-Java Thread Safety

- Subjects are not thread safe.

RX-Java Thread Safety

- Subjects are not thread safe.
- Most of the operators with single observable are not thread safe ex - take(N)

RX-Java Thread Safety

- Subjects are not thread safe.
- Most of the operators with single observable are not thread safe ex - take(N)
- Most of the operators with multiple observable are thread safe ex- merge, concat

RX-Java Thread Safety

- Subjects are not thread safe.
- Most of the operators with single observable are not thread safe ex - take(N)
- Most of the operators with multiple observable are thread safe ex- merge, concat
- Details - blog

Visibility

```
boolean run = false;
int x = 42;

public void thread1() {
  Thread thread = new Thread(new Runnable() {
    @Override public void run() {
      while (!run) { }
      System.out.println(x);
    }
  });
  thread.start();
}

public void thread2() {
  x = 100;
  run = true;
}
```

Visibility

```
boolean run = false;
int x = 42;

public void thread1() {
  Thread thread = new Thread(new Runnable() {
    @Override public void run() {
      while (!run) { }
      System.out.println(x);
    }
  });
  thread.start();
}

public void thread2() {
  x = 100;
  run = true;
}
// Output – Never terminate, 42, 100
```

Visibility

```
volatile boolean run = false;
volatile int x = 42;

public void thread1() {
  Thread thread = new Thread(new Runnable() {
    @Override public void run() {
      while (!run) { }
      System.out.println(x);
    }
  });
  thread.start();
}

public void thread2() {
  x = 100;
  run = true;
}
// Output – 100
```

WAIT & NOTIFY

PRODUCER

```
synchronized (lock) {
   sharedData = …;
   sharedDone = true;
   lock.notify();
}
```

CONSUMER

```
synchronized (lock) {
   while ( !sharedDone ) {
      lock.wait();
   }
   print(sharedData);
}
```

DOESN'T PRESERVE ORDER

PRODUCER

```
synchronized (lock) {
  sharedData = …;
  sharedDone = true;
  lock.notify();
}
```

Reorder!

```
synchronized (lock) {
  sharedDone = true;
  sharedData = …;
  lock.notify();
}
```

CONSUMER

```
synchronized (lock) {
  while ( !sharedDone ) {
    lock.wait();
  }
  print(sharedData);
}
```

SPURIOUS NOTIFICATIONS

INCORRECT CONSUMER

```
synchronized (lock) {
  if ( !sharedDone ) {
    lock.wait();
  }
  print(sharedData);
}
```

CORRECT CONSUMER

```
synchronized (lock) {
  while ( !sharedDone ) {
    lock.wait();
  }
  print(sharedData);
}
```

“Locking can guarantee both visibility and atomicity; volatile variables can only guarantee visibility.”

–Commandment #2

Put if Absent

```
public class MapExample<K, V> {

  Map<K, V> myMap = new ConcurrentHashMap<>();

  public V addIfAbsent(K key, V value) {
    if (!myMap.containsKey(key)) {
      myMap.put(key, value);
    }
    return myMap.get(key);
  }
}
```

Put if Absent

```
public class MapExample<K, V> {

  Map<K, V> myMap = new ConcurrentHashMap<>();

  public V addIfAbsent(K key, V value) {
    if (!myMap.containsKey(key)) {
      myMap.put(key, value);
    } // Non - Atomic
    return myMap.get(key);
  }
}
```

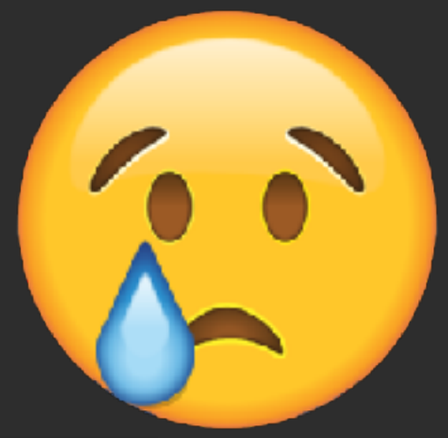
Put if Absent

```
public class MapExample<K, V> {

  Map<K, V> myMap = new ConcurrentHashMap<>();

  public synchronized V addIfAbsent(K key, V value) {
    if (!myMap.containsKey(key)) {
      myMap.put(key, value);
    }
    return myMap.get(key);
  }
}
```

Put if Absent

```
public class MapExample<K, V> {

  Map<K, V> myMap = new ConcurrentHashMap<>();

  public synchronized V addIfAbsent(K key, V value) {
    if (!myMap.containsKey(key)) {
      myMap.put(key, value);
    }
    return myMap.get(key);
  }
}

// Concurrent hash map is synchronised using another lock.
```

Put if Absent



```
public class MapExample<K, V> {

  Map<K, V> myMap = new ConcurrentHashMap<>();

  public synchronized V addIfAbsent(K key, V value) {
    synchronized (myMap) {
      if (!myMap.containsKey(key)) {
        myMap.put(key, value);
      }
      return myMap.get(key);
    }
  }
}

// Internal lock change and what if the lock is private
```

Put if Absent

```
public class MapExample<K, V> {
  MyMap<K, V> myMap = new MyMap<>(new HashMap<>());
}

class MyMap<K, V> implements Map<K, V> {

  @GuardedBy("lock") private Map<K, V> delegate;
  private Object lock = new Object();

  public MyMap(Map<K, V> map) {
    delegate = map;
  }


  @Override public V put(K key, V value) {
    synchronized (lock) {
      delegate.put(key, value);
    }
  }

  public V addIfAbsent(K key, V value) {
    synchronized (lock) {
      if (!delegate.containsKey(key)) {
        delegate.put(key, value);
      }
      return delegate.get(key);
    }
  }
}
```

Put if Absent

```
public class MapExample<K, V> {
  MyMap<K, V> myMap = new MyMap<>(new HashMap<>());
}

class MyMap<K, V> implements Map<K, V> {

  @GuardedBy("lock") private Map<K, V> delegate;
  private Object lock = new Object();

  public MyMap(Map<K, V> map) {
    delegate = map;
  }


  @Override public V put(K key, V value) {
    synchronized (lock) {
      delegate.put(key, value);
    }
  }

  public V addIfAbsent(K key, V value) {
    synchronized (lock) {
      if (!delegate.containsKey(key)) {
        delegate.put(key, value);
      }
      return delegate.get(key);
    }
  }
}
```

Put if Absent

```
public class MapExample<K, V> {
  MyMap<K, V> myMap = new MyMap<>(new HashMap<>());
}

class MyMap<K, V> implements Map<K, V> {

  @GuardedBy("lock") private Map<K, V> delegate;
  private Object lock = new Object();

  public MyMap(Map<K, V> map) {
    delegate = map;
  }

  @Override public V put(K key, V value) {
    synchronized (lock) {
      delegate.put(key, value);
    }
  }

  public V addIfAbsent(K key, V value) {
    synchronized (lock) {
      if (!delegate.containsKey(key)) {
        delegate.put(key, value);
      }
      return delegate.get(key);
    }
  }
}
```

Put if Absent

```
public class MapExample<K, V> {
  MyMap<K, V> myMap = new MyMap<>(new HashMap<>());
}

class MyMap<K, V> implements Map<K, V> {

  @GuardedBy("lock") private Map<K, V> delegate;
  private Object lock = new Object();

  public MyMap(Map<K, V> map) {
    delegate = map;
  }


  @Override public V put(K key, V value) {
    synchronized (lock) {
      delegate.put(key, value);
    }
  }

  public V addIfAbsent(K key, V value) {
    synchronized (lock) {
      if (!delegate.containsKey(key)) {
        delegate.put(key, value);
      }
      return delegate.get(key);
    }
  }
}
```

Put if Absent

```
public class MapExample<K, V> {
  MyMap<K, V> myMap = new MyMap<>(new HashMap<>());
}

class MyMap<K, V> implements Map<K, V> {

  @GuardedBy("lock") private Map<K, V> delegate;
  private Object lock = new Object();

  public MyMap(Map<K, V> map) {
    delegate = map;
  }

  @Override public V put(K key, V value) {
    synchronized (lock) {
      delegate.put(key, value);
    }
  }

  public V addIfAbsent(K key, V value) {
    synchronized (lock) {
      if (!delegate.containsKey(key)) {
        delegate.put(key, value);
      }
      return delegate.get(key);
    }
  }
}
```

Publishing

```
class ImmutableHolder<K, V> {
  private final Map<K, V> internalMap;

  ImmutableHolder(Map<K, V> internalMap) {
    this.internalMap = internalMap;
  }

  public Map<K, V> getStringMap() {
    return internalMap;
  }

  public synchronized void putValue(K key, V value) {
    internalMap.put(key, value);
  }
}
```

Publishing

```
public class Publishing {

  private ImmutableHolder<String, String> immutableHolder;

  @NotThreadSafe
  public void update() {
    immutableHolder.getStringMap().put("junk", "junk");
  }

  class ImmutableHolder<K, V> {
    private final Map<K, V> internalMap;

    ImmutableHolder(Map<K, V> internalMap) {
      this.internalMap = internalMap;
    }

    public Map<K, V> getStringMap() {
      return internalMap;
    }

    public synchronized void putValue(K key, V value) {
      internalMap.put(key, value);
    }
  }
}
```

Publishing

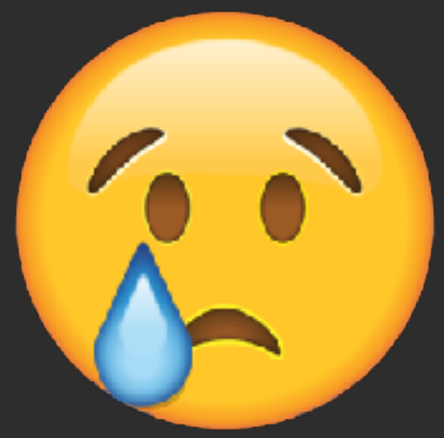
```
public class Publishing {

  private ImmutableHolder<String, String> immutableHolder;

  @NotThreadSafe
  public void update() {
    immutableHolder.getStringMap().put("junk", "junk");
  }

  class ImmutableHolder<K, V> {
    private final Map<K, V> internalMap;

    ImmutableHolder(Map<K, V> internalMap) {
      this.internalMap = internalMap;
    }

    public Map<K, V> getStringMap() {
      return internalMap.clone();
    }

    public synchronized void putValue(K key, V value) {
      internalMap.put(key, value);
    }
  }
}
```

Publishing

```
public class Publishing {

  private HashMap<String, String> mutableMap = new HashMap<>();
  private ImmutableHolder<String, String> immutableHolder = new ImmutableHolder(mutableMap);

  @NotThreadSafe
  public void update() {
    mutableMap.put("junk", "junk");
  }

  class ImmutableHolder<K, V> {
    private final Map<K, V> internalMap;

    ImmutableHolder(Map<K, V> internalMap) {
      this.internalMap = internalMap;
    }

    public Map<K, V> getStringMap() {
      return internalMap.clone();
    }

    public synchronized void putValue(K key, V value) {
      internalMap.put(key, value);
    }
  }
}
```

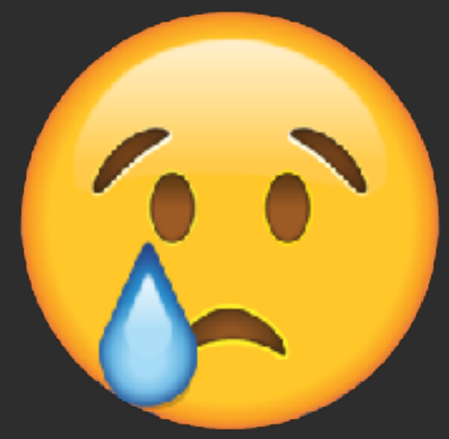
Publishing

```
public class Publishing {

  private HashMap<String, String> mutableMap = new HashMap<>();
  private ImmutableHolder<String, String> immutableHolder = new ImmutableHolder(mutableMap);

  @NotThreadSafe
  public void update() {
    mutableMap.put("junk", "junk");
  }

  class ImmutableHolder<K, V> {
    private final Map<K, V> internalMap;

    ImmutableHolder(Map<K, V> internalMap) {
      this.internalMap = internalMap.clone();
    }

    public Map<K, V> getStringMap() {
      return internalMap.clone();
    }

    public synchronized void putValue(K key, V value) {
      internalMap.put(key, value);
    }
  }
}
```



Publishing

```
public class Publishing {

  private HashMap<String, String> mutableMap = new HashMap<>();
  private ImmutableHolder<String, String> immutableHolder = new
ImmutableHolder(mutableMap);

  @NotThreadSafe
  public void update() {
    mutableMap.put("junk", “junk”); // no effect
    immutableHolder.getStringMap().put("junk", “junk"); // Crash.
  }

  class ImmutableHolder<K, V> {
    private final Map<K, V> internalMap;

    ImmutableHolder(Map<K, V> internalMap) {
      this.internalMap = Collections.unmodifiableMap(internalMap);
    }

    public Map<K, V> getStringMap() {
      return internalMap.clone();
    }
  }
}
```



Publishing

Publishing

- Immutable objects can be published anywhere.

Publishing

- Immutable objects can be published anywhere.
- Effectively immutable objects (mutable objects which we will not change) can be published using “safe publishing” - volatile, final, static initializer.

Publishing

- Immutable objects can be published anywhere.
- Effectively immutable objects (mutable objects which we will not change) can be published using “safe publishing” - volatile, final, static initializer.
- Mutable variables must be synchronized

Annotations

- `@ThreadSafe, @NotThreadSafe`
- `@Immutable`
- `@GuardedBy`
  - `@GuardedBy(“this”)`
  - `@GuardedBy(“fieldName”)`
  - `@GuardedBy(“methodName()”)`
- `@MainThread, @WorkerThread`

Summary

Summary

- Encapsulation is your best friend.

Summary

- Encapsulation is your best friend.
- Document thread safety.

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.
- Don’t let constructor escape this

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.
- Don’t let constructor escape this
- Publication:

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.
- Don’t let constructor escape this
- Publication:
    - Immutable objects can be published anywhere

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.
- Don’t let constructor escape this
- Publication:
    - Immutable objects can be published anywhere
    - Effectively immutable objects can be safely published.

Summary

- Encapsulation is your best friend.
- Document thread safety.
- Use composition to extend a synchronised class.
- Use immutable variables.
- Don’t let constructor escape this
- Publication:
    - Immutable objects can be published anywhere
    - Effectively immutable objects can be safely published.
    - Mutable object needs to be kept behind a lock.

TODO

- Java concurrency in Practice - Chapter 2-5, Chapter 13, 15, and 16.
- Devoxx talk on concurrency.
- Write code yourself?
- How to test concurrency? - JCStress

REFERENCES

JAVA CURRENCY IN PRACTICE

Brian Goetz et al

JAVA MEMORY MODEL PRAGMATICS

http://virtualjug.com/java-memory-model-pragmatics/

Aleksey Shipilëv

CLOSE ENCOUNTERS OF THE JMM KIND

https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/

Aleksey Shipilëv

Questions??